École Supérieure d'Ingénieurs
Léonard-de-Vinci

Parcours Recherche

Rapport Final

RECHERCHE COMPARATIVE DU MODELE DE COUT POUR LA NORMALISATION ET LA DENORMALISATION DE BD NOSQL

Élève :
Hao ZHANG

Mentor :
Nicolas TRAVERS

31 janvier 2019

Table des matières

1 Présentation du Sujet

Dans le BigData, les bases NoSQL répondent au problème de stockage et de distribution des données à large échelle pour souvent répondre à des problèmes de performances en centralisé.

De nombreuses solutions existent, et faire le bon choix lors de la définition des besoins d'un Système d'Information devient un choix crucial, car il est difficile et risqué de faire un changement technologique.

Le but de ce projet de recherche est de définir un modèle de coût d'évaluation de requêtes sur des solutions NoSQL.

Ainsi, ce modèle de coût (accès réseaux et accès locaux) sera capable d'estimer pour chaque type de requête ce qu'il en coûtera, et ainsi de choisir la solution NoSQL qui minimise ce coût.

L'objectif de ce projet est donc de raffiner les modèles de coût existant dans la littérature pour avoir un modèle plus global, le plus précis possible.

2 Étude Bibliographique

J'ai classifié les articles que j'ai vus pendant ce semestre en quarte groupes. La classification des articles est illustrée au-dessous(Figure 1).

I Évaluation de la Base de Donnees NoSQL

1 Évaluation des Attributs de Qualité[14]
Conception de la recherche et méthodologie :

- Identifier plusieurs attributs de qualité souhaitables à évaluer dans des bases de données NoSQL.
  - Disponibilité, cohérence, durabilité, maintenabilité, performance, fiabilité, robustesse, évolutivité, délai de stabilisation et de rétablissement
- Identifier les systèmes NoSQL les plus populaires et les plus utilisés.
  - Aerospike, Cassandra, Couchbase, CouchDB, HBase, MongoDB and Voldemort
- Explorer la littérature pour évaluer les attributs de qualité sélectionnés sur les bases de données susmentionnées.

2 Aperçu de la Performance[2] **Conception de la recherche et méthodologie :**

Comparez les cinq bases de données NoSQL les plus courantes en termes de performances des requêtes, basées sur les lectures et les mises à jour, en prenant en compte les charges de travail typiques, telles que représentées par Yahoo ! Benchmark Serving Cloud.

- Cassandra et HBase : bases de données de familles de colonnes.

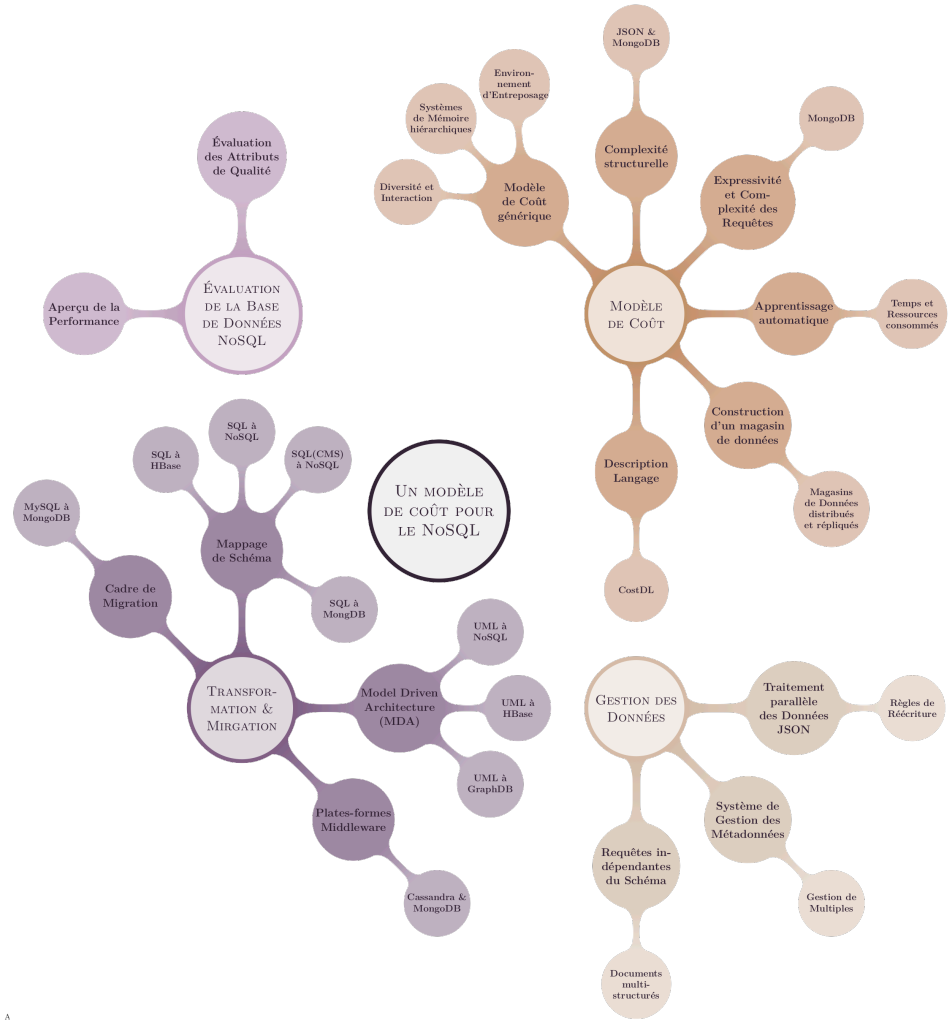

FIGURE 1 – La classification des articles

- MongoDB et OrientDB : bases de données de familles de document.
- Redis : base de données de familles de clé-valeur.

II Transformation & Mirgation

1 Transformation d'un Modele Relationnels

- Mappage de Schéma[12, 11, 7, 10]

Procédure générale :

- Phase I, Définir la base de données source.
- Phase II, Classification des tables.
- Phase III, Définition de la base de données cible.
- Phase IV, Execute Data Conversion(denormalization, migration).

Cas de recherche :
SQL à MongDB, SQL à HBase, SQL à NoSQL, SQL(CMS, Content Management System) à NoSQL

- Cadre de Migration[5]

Cadre capable de prendre en charge facilement la migration d'une base de données relationnelle (par exemple, MySQL) vers une base de données NoSQL (par exemple, MongoDB).

- Data Migration Module
  Un ensemble de méthodes permettant une migration transparente entre les SGBD (par exemple, de MySQL vers MongoDB).
- Module de Mappage de Données
  Fournir une couche de persistance pour traiter les requêtes de base de données, tout en renvoyant les données dans un format approprié.

2 Transformation d'un Modele conceptuel

- Model Driven Architecture (MDA)[13, 6, 1]

Procédure générale :

- Phase I, construire les métamodèles du diagramme de classes UML et de la base de données NoSQL.
- Phase II, proposer les règles de mappage entre les deux métamodèles.
- Phase III, construire le diagramme de classes UML, et générer le modèle de base de données NoSQL par transformation.

Cas de recherche :
UML a NoSQL, UML a HBase, UML a GraphDB

3 Plates-formes Middleware[20]

Procédure générale :

• Phase I, évaluer le surcoût de performance introduit par ces plates-formes pour les opérations CRUD.

• Phase II, comparez le coût de la migration avec et sans ces plates-formes.

Cas de recherche :

Impetus Kundera, Playorm, Spring Data

III Modèle de Coût

1 Conception physique

• Modèle de Coût générique[15, 3, 3]

Les modèles de coût conçus de ces articles concernent le modèle de coût générique surtout au niveau de la conception physique. Ces articles nous ont donnée des idées : comparer le modèle de coût des différents types de bases de données, et/ou des différentes machines de test.

• Construction d'un magasin de données[21]

Le modèle de coût conçu de cet article concerne le modèle de coût au niveau de stockage, il nous a donné une idée : comparer les entrepôts des données de différents stockages.

2 Modèle de Coût de la complexité

• Complexité structurelle[8]

• Expressivité et Complexité des Requêtes[4]

Les modèles de coût conçus de ces articles concernent le modèle de coût au niveau de la complexité. Ils nous avons donné les idées : lors de l'évaluation du modèle de coût, nous devons également envisager la mise en uvre de la complexité structurelle ou des requêtes. Aussi, comprendre bien le modèle de coût de la complexité peut nous aider à générer une prédiction pour la conception de test.

3 Apprentissage automatique[16]

Cet article nous avons donné une idée à étudier le temps d'exécution et des pensées de la conception de machine learning dans le futur.

4 Description Langage[18]

Cet article nous a proposé un métamodèle dédié à l'expression des modèles de coûts d'analyse pour les systèmes de base de données. Et notre but d'avenir c'est le tester, vérifier et l'améliorer ou le développer.

IV Gestion des Données

1 Expressivité et Complexité des Requêtes[17]

2 Traitement parallèle des Données JSON[19]

3 Requêtes indépendantes du Schéma[9]

Ces articles sont des documents complémentaires pour bien comprendre la base de données NoSQL.

Les détails de ces articles sont notés sur un autre document nommé *STATE OF THE ART*. Je l'ai ajouté à la fin de ce rapport.

3 Hypothèse

H 1 Comparer la duree pendant laquelle les bases de donnees (normalisees, semi-denormalisees ou denormalisees) de differentes tailles repondent aux quatre principaux types de commandes(CRUD), c'est-a-dire comparer leurs modeles de coût sur le temps.

H 2 A l'avenir, sur la base des resultats obtenus, comparez les theories existantes et decouvrez les lois mathematiques.

4 Démarche Scientifique

4.1 Modélisation de la base de données

4.1.1 Normalisation

La normalisation de base de données est le processus de structuration d'une base de données relationnelle selon une série de formes dites normales afin de réduire la redondance des données et d'améliorer leur intégrité.

La normalisation implique l'organisation des colonnes (attributs) et des tables (relations) d'une base de données afin de s'assurer que leurs dépendances sont correctement appliquées par les contraintes d'intégrité de la base de données. Pour ce faire, certaines règles formelles sont appliquées, soit par un processus de synthèse (création d'une nouvelle conception de base de données), soit par décomposition (amélioration d'une conception de base de données existante).

Le but essentiel de la normalisation est d'éviter les anomalies transactionnelles pouvant découler d'une mauvaise modélisation des données et ainsi éviter un certain nombre de problèmes potentiels tels que les anomalies de lecture, les anomalies d'écriture, la redondance des données et la contre-performance.

4.1.2 Dénormalisation

La dénormalisation est une stratégie utilisée sur une base de données préalablement normalisée pour améliorer les performances. En informatique, la dénormalisation consiste à essayer d'améliorer les performances de lecture d'une base de données, au prix de pertes d'écriture, en ajoutant des copies redondantes de données ou en regroupant des données. Il est souvent motivé par les performances ou l'évolutivité des logiciels de bases de données relationnelles nécessitant d'effectuer un très grand nombre d'opérations de lecture. La dénormalisation ne doit pas être confondue avec la forme non normalisée. Les bases de données / tables doivent d'abord être normalisées pour être efficacement dénormalisées.

La dénormalisation peut être définie comme la copie des mêmes données dans plusieurs documents ou tables afin de simplifier / optimiser le traitement des requêtes ou d'adapter les données de l'utilisateur à un modèle de données particulier. Elle permet de stocker des données dans une structure adaptée pour simplifier le traitement des requêtes. L'objectif pour les systèmes NoSQL est de proposer un schéma de stockage permettant d'éviter l'utilisation des jointures pour lesquelles les systèmes NoSQL sont peu adaptés en raison de l'architecture distribuée sous-jacente.

Systèmes applicables : base de données clé-valeur, bases de données document, base de données orientée colonnes et base de données orientée documents.

4.2 Base de données du test

4.2.1 Au niveau de volume...

Basé sur l'objectif de comparer différents modèles de coût pouvant être générés par le même type d'instructions sur les différents volumes d'entrepôts de données, l'entrepôt de données de cette expérience est également conçu en tenant compte de la diversité des capacités. Nous utilisons N pour numéroter le nombre d'entrepôts de données. Le nombre total d'entrepôts de données dans cette expérience est : $N \in \{1, \quad 10, \quad 20, \quad 30, \ldots, 100\}$.

Cependant, en raison des limitations de l'ordinateur personnel de l'opérateur et du temps limité, je n'ai pris que la limite supérieure de 40 pour N dans cette expérience.

4.2.2 Collections normalisées originales

Dans cette étude, la base de données utilisée est une base de données d'une plateforme d'achat en ligne simulée. La structure de chaque de celle-ci est illustrée au-dessous(Figure 2). La combinaison de base de données entière est divisée en neuf collections : Warehous, District, Customer, Order, Order-Line, etc.

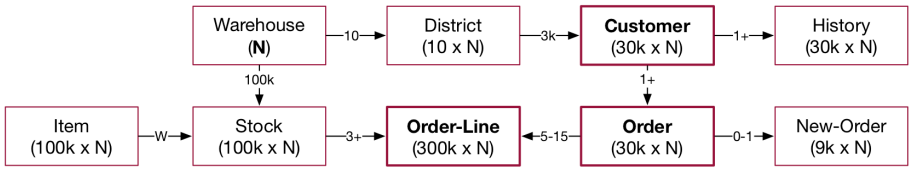

FIGURE 2 – La combinaison de base de données entière

Lorsque nous utilisons N entrepôts de données comme objets de recherche, chaque objet étant divisé en 10 blocs, donc le nombre total de blocs que nous étudions à ce moment est de $N \times 10$. Pour chaque bloc, il y a normalement 3000 clients. Les clients sont totalement différents entre les blocs

et les entrepôts. Pour chaque bloc, il y a 3000 commandes. Les commandes sont aussi totalement différentes entre les blocs et les entrepôts. En plus, chaque commande contient de 5 à 15 d'articles.

C'est-à-dire, lorsque nous utilisons N entrepôts de données comme objets de recherche, chaque objet contient $30000 \times N$ clients, $30000 \times N$ de commandes et plus de $300000 \times N$ d'articles.

Au niveau de dénormalisation, ces collections, sauf les collections de Warhous et District, ont regroupé les données pour les différents entrepôts et blocs, pour que dans chaque collection, les identités des valeurs puissent être distinguées par différents entrepôts et leurs blocs.

Mais pour une base de données Nosql, ce degré de dénormalisation ne répond pas aux exigences de conception de la base de données Nosql pour optimiser les performances des lectures. Ces collections, en effet, semblent même un peu normalisées.

Par conséquent, on prend les collections Customer, Order et Order_Line comme l'étude de la normalisation dans mon expérience.

4.2.3 Collection semi-dénormalisée

En fait, la collection semi-dénormalisée auquel il est fait référence a satisfait aux exigences de dénormalisation dans une certaine mesure, pas vraiment un nom officiel, mais seulement pour distinguer d'une collection d'un degré de dénormalisation plus élevé dans cette expérience.

La structure de cette collection semi-dénormalisée est montrée au-dessous(Table 1).

Semi-Dénormalisation	
PK	o_id
	o_d_id
	o_w_d
	o_c_id
	⋮
	order_lines
	ol_id
	ol_d_id
	ol_w_d
	⋮

TABLE 1 – La structure de la collection semi-dénormalisée

En bref, pour l'étude de semi-dénormalisation, on copie la collection Order, plus crée un attriubeun comme un sous-document nommé order_lines

pour stocker toutes les informations de la collection Order_Line.

Sur la base des caractéristiques de la base de données Nosql, nous nous attendons que notre collection de Semi-Dénormalisation soit plus efficace que les collections normalisées lors de la cherche des informations détaillées pour chaque commande (avec l'information de tous les articles).

4.2.4 Collection dénormalisée

Étant donné que je n'ai pris que trois collections normalisées pour cette expérience, on peut comprendre cette collection dénormalisée d'ici comme une collection ayant un degré de démoralisation complet.

La structure de cette collection dénormalisée est montrée au-dessous(Table 2).

Dénormalisation	
PK	c_id
	c_d_id
	c_w_d
	⋮
	customer_order
	o_id
	o_d_id
	o_w_d
	o_c_id
	⋮
	order_lines
	ol_id
	ol_d_id
	ol_w_d
	⋮

TABLE 2 – La structure de la collection dénormalisée

De la même façon, pour l'étude de dénormalisation, on copie la collection Customer, plus crée un sous-document nommé customer_order pour stocker toutes les informations de la collection Semi-Dénormalisation. Sa structure est au-dessous.

Sur la base des caractéristiques de la base de données Nosql, nous nous attendons que notre collection de Dénormalisation soit plus efficace que les collections normalisées et la collection semi-dénormalisée lors de la cherche

des informations détaillées pour chaque client (avec l'information de toutes leurs commandes et les articles de chaque commande).

4.3 Choix de techniques

Afin d'atteindre notre but, j'ai choisi Python 3.7 comme langue principale pour manipuler ma recherche. Grâce à l'API PyMongo, Python peut exécuter une grande partie des fonctions même que MongoDB même avec des différences distinctes à l'issue de la nature de langage. Bien qu'il soit possible d'exécuter MongoDB sans Python, je voudrais essayer de construire un programme qui peut l'exécuter automatiquement et donner des résultats visualisés.

Les codes principaux de connexion et la modélisation sont dans l'annexe.

Lors du processus de modélisation, je stocke d'abord les résultats de la modélisation dans un fichier json, puis réimporte le fichier json dans une nouvelle base de données, ce qui évite la possibilité que les deux programmes de modélisation et de construction de la base de données soient exécutés simultanément. Conflit

Afin d'accélérer le temps consacré à la modélisation, selon la suggestion de l'enseignant, j'ai ajouté des index aux identités de chaque collection (o_id, o_c_id par example), identités de chaque entrepôt (o_w_id par example) et identités de chaque bloc (o_d_id par example). L'effet d'accélération est évident.

4.4 Requêtes des tests

L'un des objectifs de ce test consiste également à étudier le temps de réponse d'entrepôts de différentes tailles pour quatre instructions différentes, à savoir la consommation de processus.

Dans le but d'examiner les quatre propriétés de CRUD (create, read, update, delete) , les requêtes conçues sont à peu près les suivantes :

C Create : creer une nouvelle commande avec 10 articles differents

R Read : chercher aleatoirement l'information d'un client par son nom et prenom en montrant l'id de la plus nouvelle commande

U Update : modifier le temps de livraison des dix articles les plus anciens qui ne sont pas encore delivres

D Delete : supprimer aleatoirement d'une commande pas encore livree

Pour chaque requête, j'ai exécuté vingt fois par le groupe des collections normalisées, semi-dénormalisée et dénormalisée pour un seul entrepôt, dix entrepôts, vingt entrepôts, trente entrepôts et quarante entrepôts. À cause

du temps limité et la durée longue d'installation et des tests, je suis arrivé seulement à installer et à tester pour quarante entrepôts au maximum.

Pour chaque fois de test, j'ai noté les temps de procédure par la fonction `time.process_time()` (avec l'unit d'heure) du package `time` de Python. Pour la visualisation des résultats, j'ai choisi de conserver les temps de processus dans un DataFrame. Et les requêtes sur chaque modalisation sont distinguées par les colonnes de ce DataFrame.

Les codes principaux des quatre requêtes sont aussi dans l'annexe.

5 Interpretation des Resultats

5.1 Visualisation des Résultats

Pour bien visualiser les résultats, j'ai choisi de montrer les résultats en deux 's différentes.

La première figure des résultats(Figure 3) a montré les résultats de temps moyen avec l'écart type. Pour les résultats des requêtes différentes. Les résultats sont regroupés en quartes sous-figures dedans. Ils se distinguent par quatre nuances différentes de blue, rose, orange et vert.

Dans chaque sous-figure, les requêtes des modélisations différentes sont présentées par les barres différentes. Et les barres sont regroupées en cinq groups par les résultats d'un entrepôt, dix entrepôts, vingt entrepôts, trente entrepôts et quarante entrepôts.

Toutes les barres colorées présentent les résultats moyens des vingt fois de tests, et les lignes verticales présentent les résultats d'écart type.

Attention ! Afin de mieux présenter les résultats de chaque sous-figure, les maximums de chacun sont différents.

La deuxième figure des résultats(Figure 4) a montré les résultats de temps par chaque fois de test. Pour les résultats des requêtes différentes. Les résultats sont regroupés en quartes sous-figures dedans. Dans les sous-figures, les requêtes des trois modélisations se distinguent par trois couleurs : bleu, violet, red.

Différents de la figure précédente, tous les résultats sont notés dans cette figure, et sont divisés par les lignes verticales de 0, 20, 40, 60, 80 pour distinguer les vingt fois de tests pour un entrepôt, dix entrepôts, vingt entrepôts, trente entrepôts et quarante entrepôts.

5.2 Observation et supposition

Pour requête C, il est évident que le temps de procédé de la collection semi-dénormalisée est plus court que les autres collections. Mais les résultats sont peut-être liens à nature des instructions d'exécution de chaque collection. Dans la collection semi-dénormalisée, l'instruction d'exécution de

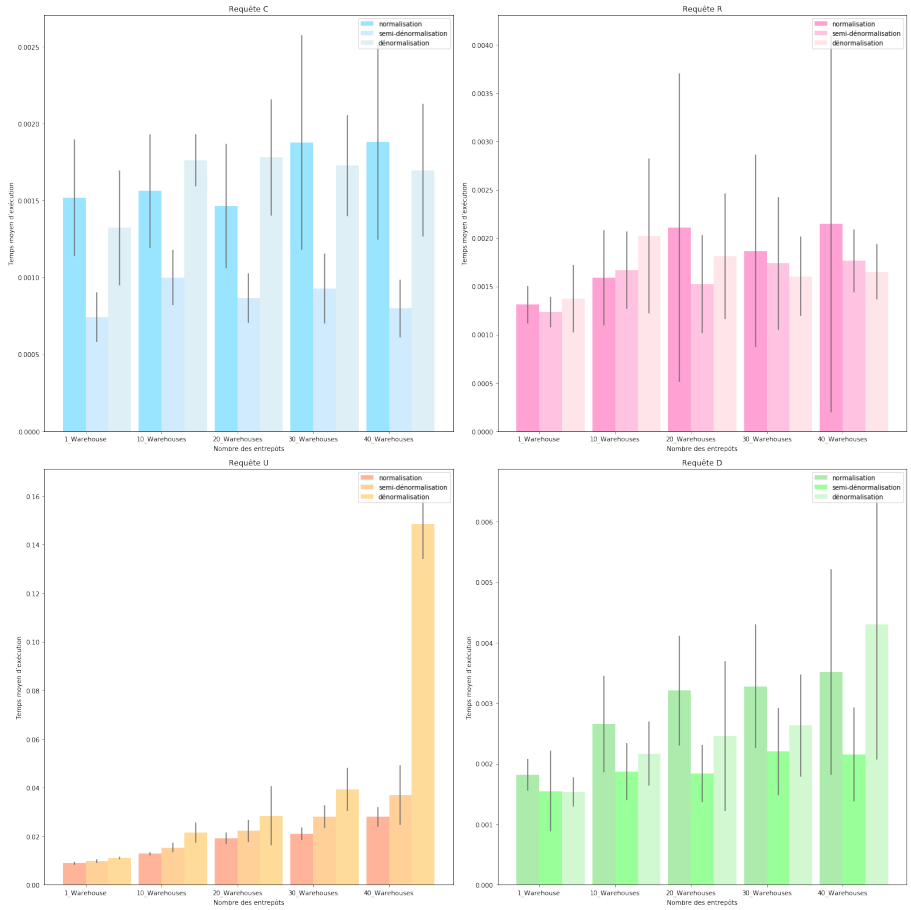

FIGURE 3 – Resultats de temps moyen avec l'ecart type

FIGURE 4 – Resultats des tests par chaque fois de test

MongoDB est insertOne(). Mais dans les collections normalisées, les instructions d'exécution sont insertOne() et insertMany(). Pour la collection dénormalisée, car on ne crée pas un nouveau client, l'instruction d'exécution devient update(), elle est plutôt comme une requête de mise à jour.

Pour requête R, on a vu qu'à mesure que la capacité augmente, le temps de processus de la collection dénormalisée a un net avantage sur les collections normalisées après un volume supérieur de 20 entrepôts. C'est très intéressant pour faire des tests sur des bases de données plus grandes de 40 entrepôts à voir si toujours la requête R fonctionne efficacement dans une collection dénormalisée.

Les résultats de la requête U sont toujours très haut à comparer d'autres requêtes, même dans les collections normalisées. Mais, le temps de processus de la collection dénormalisée est énormément argumenté entre les trente entrepôts et quarante entrepôts. Cela signifie-t-il qu'une valeur critique de la mémoire de l'ordinateur a été atteinte ? C'est une question qui mérite d'être vérifiée et étudiée à plusieurs reprises.

Pour la requête U et la requête D de la dénormalisation, j'ai utilisé l'instruction d'exécution d'update() pour réaliser ces requêtes spécialisées. Évidemment, cela manque la comparabilité des résultats. Entre les trente entrepôts et quarante entrepôts, le temps de processus n'a pas encore rencontré une augmentation énorme. Évidemment, les résultats de requête D ressemblent aux résultats de requête R. Et la collection semi-normalisée a eu toujours la meilleure efficacité.

6 Conclusion

A cause de temps limite, la recherche actuelle sur le modele de cout de la base de donnees NoSQL est encore au stade initial de la modelisation. Au cours du processus, j'ai rencontre de nombreux problemes, tels que le temps de fusion de la base de donnees est trop long ou le langage de requete n'est pas assez precis. Une fois la modelisation actuelle mise au point, il est necessaire de repeter les pratiques pour obtenir un resultat de test plus credible.Quant a savoir si nous pouvons enfin trouver une loi credible, nous devons la tester tout le temps.

Bien que cette étude expérimentale ne soit que le début de la modélisation du modèle de coût, mais les résultats des tests des quatre requêtes obtenus conformes également aux attentes et des caractéristiques structurelles d'une base des données NoSQL. Il est très important de continuer ces tests avec une base de données d'un volume plus grande comme la conception. Aussi, pour mieux tester les bases des données, il vaut mieux varier les requêtes de tests, surtout les requêtes C(Create) et R(Read).

7 Perspective

Pour un sujet axé sur la recherche, sa tâche ne doit pas se limiter à des tests, mais en être déduite. Par conséquent, pour ce sujet, lorsque les résultats des tests de tous les cent entrepôts de données sont établis, il est naturel de combiner la littérature scientifique pour en déduire des lois et même des formules de données. Les facteurs affectant cette formule peuvent être nombreux, tels que les performances de la machine ou la complexité de la structure.

De même, pour la recherche dans la base de données NoSQL, nous pouvons non seulement nous arrêter à MongoDB, mais également nous étendre progressivement au même type de base de données(bases de données document), voire à différents types de bases de données.

8 Annexe

8.1 Code de la connexion de MongoDB

```python
from pymongo import MongoClient

client = MongoClient(host="localhost", port=27017)
db = client["1_Warehouse"]
```

8.2 Code principal de la semi-dénormalisation

```
ujson.dumps(
db.Order_1.aggregate(
[{
  "$match": {
    "o_id": i_str
  }
},
{
  "$lookup": {
    "from":
    "Order_Line_1",
    "let": {
      "o_id": "$o_id",
      "o_d_id": "$o_d_id",
      "o_w_id": "$o_w_id"
    },
    "pipeline": [{
      "$match": {
        "$expr": {
          "$and": [{
            "$eq": ["$ol_id", "$$o_id"]
          }, {
            "$eq": ["$ol_d_id", "$$o_d_id"]
          }, {
            "$eq": ["$ol_w_id", "$$o_w_id"]
          }]
        }
      }
    }],
    "as":
    "order_lines"
  }
}, {
  "$out": "tmp"
}]))
```

8.3 Code principal de la dénormalisation

```
ujson.dumps(
db.Customer_1.aggregate(
[{
  "$match": {
    "c_id": i_str
  }
},
{
  "$lookup": {
    "from":
    "Semi_1",
    "let": {
      "c_id": "$c_id",
      "c_d_id": "$c_d_id",
      "c_w_id": "$c_w_id"
    },
    "pipeline": [{
      "$match": {
        "$expr": {
          "$and": [{
            "$eq": ["$o_c_id", "$$c_id"]
          }, {
            "$eq": ["$o_d_id", "$$c_d_id"]
          }, {
            "$eq": ["$o_w_id", "$$c_w_id"]
          }]
        }
      }
    }],
    "as":
    "customer_order"
  }
}, {
  "$out": "tmp"
}]))
```

8.4 Code principal de la requête C

```
Order_1.insert_one({
  "o_id": "4001",
  "o_d_id": district,
  "o_w_id": warehouse,
  "o_c_id": customer,
  "o_entry_d": daytime,

})

Order_Line_1.insert_many([{
```

```
    "ol_id": "4001",
    "ol_d_id": district,
    "ol_w_id": warehouse,
    "ol_delivery_d": daytime,

},

])

order_lines = [{
    "ol_id": "4001",
    "ol_d_id": district,
    "ol_w_id": warehouse,
    "ol_delivery_d": daytime,

},

]

Semi_1.insert_one({
    "o_id": "4001",
    "o_d_id": district,
    "o_w_id": warehouse,
    "o_c_id": customer,
    "o_entry_d": daytime,

    "order_lines": order_lines
})

customer_order = [{
    "o_id": "4001",
    "o_d_id": district,
    "o_w_id": warehouse,
    "o_c_id": customer,
    "o_entry_d": daytime,

    "order_lines": order_lines
}]

Denormal_1.update_one({
    "c_id": customer,
    "c_d_id": district,
    "c_w_id": warehouse
}, {
    "$set": {
    }
},
upsert=True)
```

8.5 Code principal de la requête R

```
Customer_id = Customer_1.find_one({
  "$and": [{
    "c_first": Customer_Info.get("c_first")
  }, {
    "c_last": Customer_Info.get("c_last")
  }]
}, {
  "_id": 0,
  "c_id": 1,
  "c_d_id": 1,
  "c_w_id": 1
})

Order_1.aggregate([{
  "$match": {
    "$and": [{
      "o_c_id": Customer_id.get("c_id")
    }, {
      "o_d_id": Customer_id.get("c_d_id")
    }, {
      "o_w_id": Customer_id.get("c_w_id")
    }]
  }
}, {
  "$sort": {
    "o_entry_d": -1
  }
}, {
  "$limit": 1
}, {
  "$project": {
    "_id": 0,
    "o_id": 1
  }
}])


Semi_1.aggregate([{
  "$match": {
    "$and": [{
      "o_c_id": Customer_id.get("c_id")
    }, {
      "o_d_id": Customer_id.get("c_d_id")
    }, {
      "o_w_id": Customer_id.get("c_w_id")
    }]
  }
}, {
  "$sort": {
    "o_entry_d": -1
  }
}, {
  "$limit": 1
```

```
}, {
  "$project": {
    "_id": 0,
    "o_id": 1
  }
}])


Denormal_1.aggregate([{
  "$match": {
    "$and": [{
      "c_id": Customer_id.get("c_id")
    }, {
      "c_d_id": Customer_id.get("c_d_id")
    }, {
      "c_w_id": Customer_id.get("c_w_id")
    }]
  }
}, {
  "$sort": {
    "customer_order.o_entry_d": -1
  }
}, {
  "$limit": 1
}, {
  "$project": {
    "_id": 0,
    "customer_order.o_id": 1
  }
}])
```

8.6 Code principal de la requête U

```
Order_Line_1.find_one_and_update({
  "ol_delivery_d": None
}, {"$set": {
    "ol_delivery_d": daytime
}},
sort=[("ol_id", pymongo.ASCENDING)])

Semi_1.find_one_and_update({
  "order_lines.ol_delivery_d": None
}, {"$set": {
    "order_lines.$.ol_delivery_d": daytime
}},
sort=[("order_lines.ol_id", pymongo.ASCENDING)])

Denormal_1.find_one_and_update(
{
  "customer_order.order_lines.ol_delivery_d": None
```

```
}, {"$set": {
    "customer_order.0.order_lines.1.ol_delivery_d": daytime
}},
sort=[("customer_order.order_lines.ol_id", pymongo.ASCENDING)])
```

8.7 Code principal de la requête D

```
Order_1.delete_one({
  "$and": [{
    "o_id": Order_Line_Info.get("ol_id")
  }, {
    "o_d_id": district
  }, {
    "o_w_id": warehouse
  }]
})

Order_1.delete_many({
  "$and": [{
    "ol_id": Order_Line_Info.get("ol_id")
  }, {
    "ol_d_id": district
  }, {
    "ol_w_id": warehouse
  }]
})

Semi_1.delete_one({
  "$and": [{
    "o_id": Order_Line_Info.get("ol_id")
  }, {
    "o_d_id": district
  }, {
    "o_w_id": warehouse
  }]
})

Denormal_1.update_one({
  "customer_order.o_id": Order_Line_Info.get("ol_id"),
  "customer_order.o_d_id": district,
  "customer_order.o_w_id": warehouse
}, {
  "$unset": {
    "customer_order._id": "",
    "customer_order.o_id": "",
    "customer_order.o_d_id": "",
    "customer_order.o_w_id": "",
    "customer_order.o_c_id": "",
    "customer_order.o_entry_d": "",

```

```
44     }
45 },
46 upsert=True)
```

Reference

[1] Fatma Abdelhedi, Amal Ait Brahim, Faten Atigui, and Gilles Zurfluh. Mda-based approach for nosql databases modelling. pages 88–102, 08 2017.

[2] Veronika Abramova, Jorge Bernardino, and Pedro Nuno San-Bento Furtado. Which nosql database ? a performance overview. *OJDB*, 1 :17–24, 2014.

[3] Ladjel Bellatreche, Salmi Cheikh, Sebastian Breß, Amira Kerkad, Ahcène Boukorca, and Jalil Boukhobza. How to exploit the device diversity and database interaction to propose a generic cost model ? In *IDEAS*, 2013.

[4] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, and Guohui Xiao. Expressivity and complexity of mongodb queries. In *Proc. of the 21st Int. Conf. on Database Theory (ICDT 2018)*, volume 98 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9 :1–9 :23. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[5] Leonardo C. da Rocha, Fernando Vale, Elder Cirilo, Dárlinton Barbosa, and Fernando Mourão. A framework for migrating relational datasets to nosql 1 . In *ICCS*, 2015.

[6] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Umltographdb : Mapping conceptual schemas to graph databases. In *ER*, 2016.

[7] Myller Claudino de Freitas, Damires Yluska Souza, and Ana Carolina Salgado. Conceptual mappings to convert relational into nosql databases. In *ICEIS*, 2016.

[8] Paola Gómez, Claudia Roncancio, and Rubby Casallas. *Towards Quality Analysis for Document Oriented Bases : 37th International Conference, ER 2018, Xi'an, China, October 2225, 2018, Proceedings*, pages 200–216. 09 2018.

[9] Hamdi Ben Hamadou, Faïza Ghozzi, André Péninou, and Olivier Teste. Towards schema-independent querying on document data stores. In *DOLAP*, 2018.

[10] Shady Hamouda and Zurinahni Zainol. Document-oriented data schema for relational database migration to nosql. *2017 International Conference on Big Data Innovations and Applications (Innovate-Data)*, pages 43–50, 2017.

[11] Chao-Hsien Lee and Yu-Lin Zheng. Sql-to-nosql schema denormalization and migration : A study on content management systems. *2015 IEEE International Conference on Systems, Man, and Cybernetics*, pages 2022–2026, 2015.

[12] Chongxin Li. Transforming relational database into hbase : A case study. *2010 IEEE International Conference on Software Engineering and Service Sciences*, pages 683–687, 2010.

[13] Yan Li, Ping Gu, and Chao Zhang. Transforming uml class diagrams into hbase based on meta-model. *2014 International Conference on Information Science, Electronics and Electrical Engineering*, 2 :720–724, 2014.

[14] João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right nosql database for the job : a quality attribute evaluation. *J. Big Data*, 2 :18, 2015.

[15] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, 2002.

[16] Andréa M. Matsunaga and José A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 495–504, 2010.

[17] Sergi Nadal, Alberto Abelló, Oscar Romero, Stijn Vansummeren, and Panos Vassiliadis. Mdm : Governing evolution in big data ecosystems. In *EDBT*, 2018.

[18] Abdelkader Ouared, Yassine Ouhammou, and Ladjel Bellatreche. Costdl : A cost models description language for performance metrics in database. *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 187–190, 2016.

[19] Christina Pavlopoulou, E. Preston Carman, Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. A parallel and scalable processor for json data. In *EDBT*, 2018.

[20] Ansar Rafique, Dimitri Van Landuyt, Bert Lagaisse, and Wouter Joosen. On the performance impact of data access middleware for nosql data stores a study of the trade-off between performance and migration cost. *IEEE Transactions on Cloud Computing*, 6 :843–856, 2018.

[21] Heinz Stockinger, Kurt Stockinger, Erich Schikuta, and Ian Willers. Towards a cost model for distributed and replicated data stores. In *PDP*, 2001.

ESILV, École supérieure d'ingénieurs Léonard-de-Vinci
Subject: Un modèle de coût pour le NoSQL
Professor: Nicolas TRAVERS

DVRC, De Vinci Research Center
Discipline: NOSQL Databases
Student: Hao ZHANG

STATE OF THE ART

1 Évaluation de la Base de Donnees NoSQL

1.1 Évaluation des Attributs de Qualité

(1) **Choosing the right NoSQL database for the job: a quality attribute evaluation**

(a) **Keywords:** NoSQL databases; Key-value; Document store; Columnar; Graph; Software engineering; Quality attributes; Software architecture.

(b) **Aim:** Survey and create a concise and up-to-date comparison of NoSQL engines, identifying their most beneficial use case scenarios from the software engineer point of view and the quality attributes that each of them is most suited to.

(c) **Research design and methodology:**

i) Identifying several desirable quality attributes to evaluate in NoSQL databases.
ii) Identifying which NoSQL systems are more popular and used.
iii) Surveying the literature to evaluate the selected quality attributes on the aforementioned databases.

(d) **Evaluated NoSQL databases:** Aerospike, Cassandra, Couchbase, CouchDB, HBase, MongoDB and Voldemort.

(e) **Software quality attributes:** Availability, Consistency, Durability, Maintainability, Performance, Reliability, Robustness, Scalability, Stabilization Time and Recovery Time.

(f) **Summary for popular databases**

i) Aerospike suffers from data loss issues, affecting its durability, and it also has issues with stabilization time (in particular in synchronous mode).
ii) Cassandra is a multi-purpose database (in particular due to its configurable consistency properties) which mostly lacks read performance (since it is tuned for write-heavy workloads).
iii) CouchDB provides similar technology to MongoDB, but is better suited for situations where availability is needed.
iv) Couchbase provides good availability capabilities (coupled with good recovery and stabilization times), making it a good candidate for situations where failover is bound to happen.
v) HBase has similar capabilities to Cassandra, but is unable to cope with high loads, limiting its availability in these scenarios, and is also the worst database in terms of robustness.
vi) MongoDB is the database that mostly resembles the classical relational use case scenario - it is better suited for reliable, durable, consistent and read-oriented use cases. It is somewhat lacking in terms of availability, scalability, and write-performance, and it is very hindered by its stabilization time (which is also one of the reasons for its low availability). Furthermore, it is not as efficient during write operations.
vii) Although lack some information on Voldemort, find it to be a poor pick in terms of maintainability. It is, however, a good pick for durable, write-heavy scenarios, and provides a good balance between read and write performance.

1.2 Aperçu de la Performance

(2) **Which NoSQL Database? A Performance Overview**

(a) **Keywords:** NoSQL databases, performance evaluation, execution time, benchmark, YCSB(*Yahoo! Cloud Serving Benchmark*)

(b) **Aim:** Compare the five most popular NoSQL databases in terms of query performance, based on reads and updates, taking into consideration the typical workloads, as represented by the Yahoo! Cloud Serving Benchmark.

(c) **NOSQL DATABASES:**

i) Cassandra and HBase: Column Family databases.

ii) MongoDB and OrientDB: Document Store databases.

iii) Redis: Key-value Store database.

(d) **EXPERIMENTAL SETUP:**

i) This benchmark of YCSB (Yahoo! Cloud Serving Benchmark, which allows to evaluate and compare the performance of NoSQL databases) consists of two components: a data generator and a set of performance tests consisting, in a simplistic way, of *read* and *insert* operations.

ii) Only execute workloads A(**50% read and 50% update**), C(**100% read**) and an additional workload H, defined by the user, which is **100% update**.

iii) In order to evaluate the databases, randomly generate 600,000 records, each with 10 fields of 100 bytes over the key registry identification, resulting in roughly 1kb total per record, while varying the number of stored records and operations.

(e) **Overall Evaluation:**

i) The overall results show that the in-memory database, Redis, had the best performance. Redis is a Key-value Store database and is highly optimized for performing *get* and *put* operations due to mapping data into RAM.

ii) Cassandra and HBase, as Column Family databases, showed good update performance, since they are optimized for update operations.

iii) Document Store databases had the worst execution times, and OrientDB is the database with the lowest overall performance.

2 Transformation & Mirgation

2.1 Model Driven Architecture (MDA)

(3) **Transforming UML Class Diagrams into HBase Based on Meta-model**

(a) Keywords:
HBase; MDA; meta-model; NoSQL; model transformation

(b) Aim:
Based on the ideas of Model Driven Architecture (MDA), propose a method that transforms UML class diagrams into HBase based on Meta-model.

(c) Achievement:
Propose the mapping rules of every element of the meta-model of UML class diagram and HBase database model to realize the transformation from class diagram to HBase data model after building the meta-model of UML class diagram and HBase database model.

(d) Procedure:

i) *Phase I*, the meta-models of UML class diagram and HBase database are built.

ii) *Phase II*, the mapping rules between the two meta-models are proposed.

iii) *Phase III*, the UML class diagram is built and the HBase database model is generated by transformation.

(e) Evaluation:
This paper uses Atlas language to achieve a breakfast serving system to prove the feasibility of the MDA in the software development.

(4) **UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases**

(a) Keywords:
Database Design; UML; OCL; NoSQL; Graph Database; Gremlin

(b) Aim:
Describe a mapping from UML/OCL conceptual schemas to Blueprints and Gremlin via an intermediate Graph metamodel.

(c) Reference:

i) *Blueprints*, an abstraction layer on top of a variety of graph databases.

ii) *Gremlin*, a graph traversal language.

iii) *Graph metamodel*, an intermediate representation to facilitate the integration of several kinds of graph databases.

(d) Achievement:

i) UMLtoGraphDB has been implemented as a collection of open-source Eclipse plugins, available on Github.

ii) UMLtoGraphDB takes as input the UML and OCL files that are then translated, respectively, by the Class2GraphDB and OCL2Gremlin ATL transformations.

iii) These transformations add up to a total of 110 rules and helper functions.

(e) Procedure:

i) *Phase I*, translate conceptual schemas expressed using the Unified Modeling Language (UML) into a graph representation.

ii) *Phase II*, generate database-level queries from business rules and invariants defined using the Object Constraint Language (OCL).

5 **MDA-based approach for NoSQL Databases modelling**

(a) Keywords:
UML; NoSQL; Big Data; MDA; QVT; Models Transformation

(b) Aim:
Propose an automatic MDA-based approach that provides a set of transformations, formalized with the QVT language, to translate UML conceptual models into NoSQL models.

(c) Reference:

i) *MDA*, Model Driven Architecture.

ii) *QVT*, Query / View / Transformation.

(d) Achievement:
Build an intermediate logical model compatible with column, document and graph-oriented systems.

(e) Procedure:

i) *Phase I*, the UMLtoGenericModel transformation – define the source (UML Class Diagram) and the target (Generic Logical Model).

ii) *Phase II*, the GenericModeltoPhysicalModel transformation – creates NoSQL physical models starting from the proposed generic logical model.

(f) Advantage:
This model remains stable, even though the NoSQL system evolves over time which simplifies the transformation process and saves developers efforts and time.

2.2 Mappage de Schéma

6 **Transforming Relational Database into HBase: A Case Study**

(a) Keywords:
HBase; data transformation; schema mapping

(b) Aim:
Propose a novel approach that transforms a relational database into HBase.

(c) Achievement:
A case study is employed to demonstrate this proposed solution.

(d) Procedure:

i) *Phase I*, relational schema is transformed into HBase schema based on the data model of HBase. With three guidelines:

- Group correlated data in a column family.
- For each side of a relationship, add the foreign key references of the other side if it needs to access the other side's objects.
- Merge attached data tables to reduce foreign keys.

ii) *Phase II*, relationships between two schemas are expressed as a set of nested schema mappings, which would be employed to create a set of queries or programs that transform the source relational data into the target representation automatically.

7 **SQL-to-NoSQL Schema Denormalization and Migration: A Study on Content Management Systems**

(a) Keywords:
Schema Migration; Denormalization; SQL; Not Only SQL (NoSQL); Content Management System (CMS)

(b) Aim:
Propose an autonomous SQL-to-NoSQL schema denormalization and migration.

(c) Procedure:

i) *Phase I*, propose an autonomous SQL-to-NoSQL schema denormalization and migration.

ii) *Phase II*, evaluate the proposed mechanism on the realistic CMS software.

(d) Experimental Results:
This mechanism not only helps the current CMS software migrate into the cloud platform without re-designing their database schemas, but also improves at least 45% access performance after schema migration.

8 **Conceptual Mappings to Convert Relational into NoSQL Databases**

(a) Keywords:
Relational Databases; NoSQL Systems; Conceptual Mappings; Data Conversion

(b) Aim:
Propose an approach, named as R2NoSQL, which defines conceptual mappings to enhance the data conversion process.

(c) Achievement:
The approach and some implementation and experimental results, which show that, by using the defined conceptual mappings, could obtain a consistent target NoSQL database with respect to a source Relational one.

(d) Procedure:

i) *Phase I*, **Set Source Database**: it includes the definition of the relational DBMS to be used as well as the metadata and data extraction step.

ii) *Phase II*, **Classify Table**: The tables extracted from the Relational database will be classified by the user.

iii) *Phase III*, **Set Target Database**: The user chooses the target NoSQL database.

iv) *Phase IV*, **Execute Data Conversion**: It analyzes the extracted metadata and data from the source database, verifies tables' classification and possible conceptual mappings, identifies the target NoSQL concepts and persists corresponding data in the target database.

9 **Document-Oriented Data Schema for Relational Database Migration to NoSQL**

(a) Keywords:
data model; NoSQL; migration; document-oriented data

(b) Aim:
Design a schema for document-oriented data to migrate from the current traditional database schema to NoSQL.

(c) Procedure:

i) *Phase I*, propose a document-oriented data model for big data.

ii) *Phase II*, apply this model to migrate relational database applications to NoSQL on the basis of the properties of the ER model.

2.3 Cadre de Migration

10 **A Framework for Migrating Relational Datasets to NoSQL**

(a) Keywords:
Framework; Big Data Migration; Relational; NoSQL

(b) Aim:
Propose NoSQLayer, a framework capable to support conveniently migrating from relational (i.e., MySQL) to NoSQL DBMS (i.e., MongoDB).

(c) Procedure:

i) **Data Migration Module**, a set of methods enabling seamless migration between DBMSs (i.e. MySQL to MongoDB).

- *Phase I*, identify the elements of the relational database.
- *Phase II*, create a specific collection in MongoDB to store all information collected, such as table names, names of attributes, types of attributes and integrity constraints, which is called Metadata.
- *Phase III*, perform the data migration from a relational database to the NoSQL one which consists of mapping the complete requests on each table of the relational database stored in MySQL (select * from table) onto data insertions in the corresponding collections of MongoDB NoSQL database.

ii) **Data Mapping Module**, which provides a persistence layer to process database requests, being capable to translate and execute these requests in any DBMS, returning the data in a suitable format as well.

- *Phase I*, information extraction about the query.
- *Phase II*, generation and implementation of equivalent operation in NoSQL.
- *Phase III*, mapping return results.

(d) Evaluation:

i) *Qualitative Assessment*, compare results of various operations applied directly to SQL and MySQL to MongoDB using our framework.

ii) *Quantitative Assessment*, compare the runtime of different queries, varying the total number of records involved in the operations.

(e) Experimental Results:
NoSQLayer as a handful solution suitable to handle large volume of data (e.g., Web scale) in which traditional relational DBMS might be inept in the duty.

2.4 Plates-formes Middleware

11 **On the Performance Impact of Data Access Middleware for NoSQL Data Stores**

(a) Keywords:
Data management middleware; abstraction APIs; performance evaluation; migration across heterogeneous NoSQL

(b) Aim:
Propose two complementary studies for three of the most mature data access middleware platforms: Impetus Kundera, Playorm, and Spring Data to provide the following answers:

(c) Procedure:

i) *Phase I*, evaluate the performance overhead introduced by these platforms for the CRUD operations.

ii) *Phase II*, compare the cost of migration with and without these platforms.

(d) Experimental Results:

i) Despite their similarity in design, these platforms are still substantially different performance-wise.

ii) Both studies are complementary as they show the trade-off inherent in adopting a data access middleware platform for NoSQL: by allowing some performance overhead, the developer gain benefits in terms of portability and easy migration across heterogeneous data stores.

3 Modèle de Coût

3.1 Modèle de Coût générique

12 **Generic Database Cost Models for Hierarchical Memory Systems**

(a) Aim:
Propose a generic technique to create accurate cost functions for database operations.

(b) Achievement:
Aside from being useful for query optimization, the models provide insight to tune algorithms not only in a main-memory DBMS, but also in a disk-based DBMS with a large main-memory buffer cache.

(c) Procedure:

i) *Phase I*, identify a few basic memory access patterns and provide cost functions that estimate their access costs for each level of the memory hierarchy. The cost functions are parameterized to accommodate various hardware characteristics appropriately.

ii) *Phase II*, combine the basic patterns for describing the memory access patterns of database operations. The cost functions of database operations can automatically be derived by combining the basic patterns' cost functions accordingly.

(d) Experimental Results:
The experiments of DBMS prototype Monet confirm the accuracy of the cost models for different operations.

13 **How to Exploit the Device Diversity and Database Interaction to Propose a Generic Cost Model?**

(a) Keywords:
Generic Cost Model; Devices; Ontologies; Physical Design

(b) Aim:
Propose a generic cost model for the physical design that can be instantiated for each need.

(c) Procedure:

i) *Phase I*, contribute an ontology describing storage devices.

ii) *Phase II*, provide an instantiation of the meta-model for two interdependent problems: query scheduling and buffer management.

(d) Experimental Results:
The applicability of the model, as well as its effectiveness, are shown.

14 **The Generalized Physical Design Problem in Data Warehousing Environment: Towards a Generic Cost Model**

(a) Aim:
Propose the great impact of cost models along the different generations of databases.

(b) Achievement:

i) Summarize the development stages of cost models for database systems.

ii) Propose a generic cost model for the generalized physical design problem.

iii) Apply the model on the joint query scheduling and buffer management problem.

(c) Experimental Results:
The obtained results validated on Oracle11g are encouraging.

3.2 Description Langage

15 **CostDL: a Cost Models Description Language for Performance Metrics in Database**

(a) Aim:
Propose a cost models language called CostDL that allows the description of metrics related to the most sensitive database system characteristics.

(b) Achievement:
An implementation of CostDL, a language to describe database cost models, and its usage through a running example are provided.

(c) Procedure:

i) *Phase I*, formalize relevant foundations of the cost model domain.

ii) *Phase II*, suggest a meta-model dedicated to express analysis cost models for database systems.

3.3 Apprentissage automatique

16 **On the use of machine learning to predict the time and resources consumed by applications**

(a) Keywords:
application resource usage; machine learning; regression; classifier tree

(b) Aim:
Propose comparatively assesses the suitability of several machine learning techniques for predicting spatiotemporal utilization of resources by applications.

(c) Achievement:

i) Modern machine learning techniques able to handle large number of attributes are used, taking into account application- and system-specific attributes (e.g., CPU microarchitecture, size and speed of memory and storage, input data characteristics and input parameters).

ii) Extend an existing classification tree algorithm, called Predicting Query Runtime (PQR), to the regression problem by allowing the leaves of the tree to select the best regression method for each collection of data on leaves.

iii) The new method (PQR2) yields the best average percentage error, predicting execution time, memory and disk consumption for two bioinformatics applications, BLAST and RAxML, deployed on scenarios that differ in system and usage.

iv) In specific scenarios where usage is a non-linear function of system and application attributes, certain configurations of two other machine learning algorithms, Support Vector Machine and k-nearest neighbors, also yield competitive results.

(d) Experimental Results:
Experiments show that the inclusion of system performance and application-specific attributes also improves the performance of machine learning algorithms investigated.

3.4 Construction d'un magasin de données

17 **Towards a Cost Model for Distributed and Replicated Data Stores**

(a) Aim:
Propose a novel cost model for building a world-wide distributed Petabyte data store.

(b) Achievement:

i) The cost model presented is applicable for design considerations for a replicated, distributed data store.

ii) Advantages and disadvantages of a fully replicated data store are highlighted.

iii) Compare two extreme cases of data location and give a detailed discussion on partially replicated systems.

3.5 Complexité structurelle

18 **Towards quality analysis for document oriented bases**

(a) Keywords:
NoSQL; structural metrics; document-oriented systems; MongoDB

(b) Aim:
Facilitate the study of the possibilities of data structuring and providing objective metrics to better reveal the advantages and disadvantages of each solution with respect to user needs.

(c) Achievement:

i) Propose a set of structural metrics for a JSON compatible schema abstraction. These metrics reflect the complexity of the structure and are intended to be used in decision criteria for schema analysis and design process.

ii) Capitalize on experiences with MongoDB, works on XML and software complexity metrics.

3.6 Expressivité et Complexité des Requêtes

19 **Expressivity and Complexity of MongoDB Queries**

(a) Keywords:
MongoDB; NoSQL; aggregation framework; expressivity

(b) Achievement:

i) Provide a clean formal abstraction of this query language, which is called MQuery.

ii) By studying the expressivity of MQuery, show the equivalence of its well-typed fragment with nested relational algebra.

iii) By investigating the computational complexity of significant fragments of it, obtain several (tight) bounds in combined complexity, which range from LogSpace to alternating exponential-time with a polynomial number of alternations.

4 Gestion des Données

4.1 Expressivité et Complexité des Requêtes

20 **MDM: Governing Evolution in Big Data Ecosystems**

(a) Aim:
Propose the Metadata Management System, shortly MDM, a tool that supports data stewards and analysts to manage the integration and analysis of multiple heterogeneous sources under schema evolution.

(b) Achievement:

i) MDM adopts a vocabulary-based integration-oriented ontology to conceptualize the domain of interest and relies on local-as-view mappings to link it with the sources.

ii) MDM provides user-friendly mechanisms to manage the ontology and mappings.

iii) A query rewriting algorithm ensures that queries posed to the ontology are correctly resolved to the sources in the presence of multiple schema versions, a transparent process to data analysts.

(c) Experimental Results:
MDM facilitates the management of multiple can evolve data sources and enables its integrated analysis.

4.2 Traitement parallèle des Données JSON

21 **A Parallel and Scalable Processor for JSON Data**

(a) Achievement:
The rules enable new opportunities for parallelism by leveraging pipelining; they also reduce the amount of memory required as data is parsed from disk and passed up the pipeline.

(b) Procedure:

i) Introduce two categories of rewrite rules (path expression and pipelining rules) based on the JSONiq extension to the XQuery specification.

ii) Introduce a third rule category, group-by rules, that apply to both XML and JSON data.

(c) Experimental Results:
By using a large (803GB) dataset of sensor readings, the results show that the proposed rewrite rules lead to efficient and scalable parallel processing of JSON data.

4.3 Requêtes indépendantes du Schéma

22 **Towards schema-independent querying on document data stores**

(a) Aim:
Propose a novel approach to build schema-independent queries designed for querying multi-structured documents.

(b) Achievement:

i) Introduce a query enrichment mechanism that consults a pre-materialized dictionary defining all possible underlying document structures.

ii) Automate the process of query enrichment via an algorithm that rewrites select and project operators to support multi-structured documents.

(c) Experimental Results:
By conducting experiments on synthetic dataset, the results are promising when compared to the normal execution of queries on homogeneous dataset.